



✓ Programação Diferenciável (Aprendizado Profundo - UFMG)

Preâmbulo

O código abaixo consiste dos imports comuns. Além do mais, configuramos as imagens para ficar de um tamanho aceitável e criamos algumas funções auxiliares. No geral, você pode ignorar a próxima célula.

```
1 # -*- coding: utf8
2
3 import matplotlib.pyplot as plt

1 plt.rcParams['figure.figsize'] = (8, 5)
2
3 plt.rcParams['axes.axisbelow'] = True
4 plt.rcParams['axes.labelsize'] = 16
5 plt.rcParams['axes.linewidth'] = 1
6 plt.rcParams['axes.spines.bottom'] = True
7 plt.rcParams['axes.spines.left'] = True
8 plt.rcParams['axes.titlesize'] = 16
9 plt.rcParams['axes.ymargin'] = 0.1
10
11 plt.rcParams['font.family'] = 'serif'
12
13 plt.rcParams['axes.grid'] = True
14 plt.rcParams['grid.color'] = 'lightgrey'
15 plt.rcParams['grid.linewidth'] = .1
16
17 plt.rcParams['xtick.labelsize'] = 16
18 plt.rcParams['xtick.bottom'] = True
19 plt.rcParams['xtick.direction'] = 'out'
20 plt.rcParams['xtick.major.size'] = 10
21 plt.rcParams['xtick.major.width'] = 1
22 plt.rcParams['xtick.minor.size'] = 3
23 plt.rcParams['xtick.minor.width'] = .5
24 plt.rcParams['xtick.minor.visible'] = True
25
26 plt.rcParams['ytick.labelsize'] = 16
27 plt.rcParams['ytick.left'] = True
28 plt.rcParams['ytick.direction'] = 'out'
29 plt.rcParams['ytick.major.size'] = 10
30 plt.rcParams['ytick.major.width'] = 1
31 plt.rcParams['ytick.minor.size'] = 3
32 plt.rcParams['ytick.minor.width'] = .5
33 plt.rcParams['ytick.minor.visible'] = True
34
35 plt.rcParams['legend.fontsize'] = 16
36
37 plt.rcParams['lines.linewidth'] = 4
38 plt.rcParams['lines.markersize'] = 10

1 plt.style.use('tableau-colorblind10') # use um estilo colorblind!
2 plt.ion()

<contextlib.ExitStack at 0x7dc3e01cf5e0>
```

✓ 1. Tensores em Numpy

O primeiro passo para usar numpy é importar a biblioteca.

```
1 import numpy as np
```

Quando pensamos no lado prático do aprendizado profundo, um aspecto chave que ajuda na implementação de novos algoritmos é a chamada programação diferenciável. Na próxima aula vamos voltar na mesma. No momento, o importante é salientar que a programação diferenciável faz uso extensivo de Tensores.

Um [Tensor](#) é uma generalização de matrizes para mais dimensões. Quando falamos de tensores, temos três casos especiais e um genérico que engloba os outros três:

1. **Escalar:** Um tensor de zero dimensões.

```
1 1
  1
```

2. **Vetor:** Um tensor de uma dimensão.

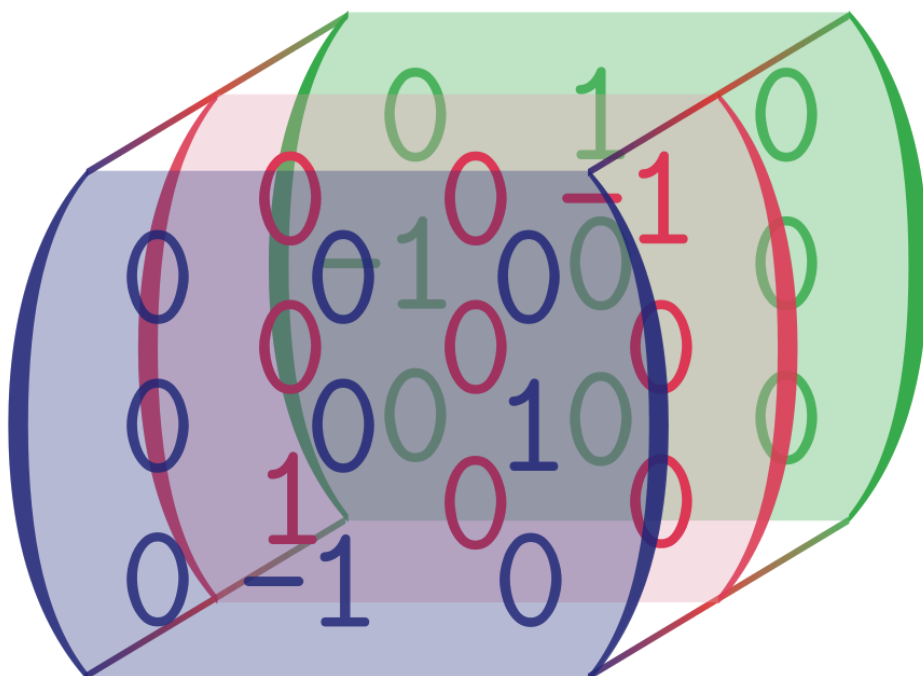
```
1 np.array([1, 2])
  array([1, 2])
```

3. **Matrizes:** Um tensor de duas dimensões.

```
1 np.array([[1, 2], [3, 4]])
  array([[1, 2],
        [3, 4]])
```

4. **Tensores.** Caso geral, representam n-dimensões. Na figura temos um tensor 3x3x3.

$$\epsilon_{ijk} =$$



▼

No exemplo abaixo, temos um tensor de dimensão $3 \times 2 \times 2$.

```
1 X = np.random.randn(3, 2, 2) # Gera números aleatórios de uma normal N(0, 1)
2 X
```

```
array([[ 0.5282071, -0.13181504],
       [-1.13237564,  0.0437249 ]],

      [[ 0.31968817, -0.26012587],
       [ 0.24790866,  0.06224302]],

      [[-0.09385319, -0.0222339 ],
       [-1.05290842,  1.27401416]]])
```

Note que ao selecionar elementos da primeira dimensão ficamos com matrizes 2×2 .

1 X[0]

```
array([[ 0.5282071, -0.13181504],
       [-1.13237564,  0.0437249 ]])
```

1 X[1]

```
array([[ 0.31968817, -0.26012587],
       [ 0.24790866,  0.06224302]])
```

1 X[2]

```
array([[-0.09385319, -0.0222339 ],
       [-1.05290842,  1.27401416]])
```

✓ 1.1) Indexando

Sendo X uma matriz:

```
1 X = np.array([[1, 2], [3, 4]])
2 X
```

```
array([[1, 2],
       [3, 4]])
```

Podemos selecionar uma linha com a sintaxe X[i], sendo i um inteiro.

1 X[0] # pegando a primeira linha de X

```
array([1, 2])
```

Podemos selecionar uma coluna com a sintaxe X[:, j], sendo j um inteiro.

1 X[:, 1] # pegando a segunda coluna de X

```
array([2, 4])
```

Podemos selecionar mais de uma linha ou coluna utilizando a sintaxe X[um_vetor] ou X[:, um_vetor], respectivamente.

```
1 X = np.array([[1, 2, 3], [4, 5, 6]])
2 X
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

1 cols = [0, 1]

2 X[:, cols] # iremos pegar a primeira e segunda colunas de X

```
array([[1, 2],
       [4, 5]])
```

Podemos selecionar linhas e colunas também indexando o tensor através de um vetor booleano.

A sintaxe `X[vetor_booleano]` retorna as linhas (ou colunas quando `X[:, vetor_booleano]`) onde o vetor é `True`.

```
1 X[[True, False]] # selecionamos apenas a primeira linha
    array([[1, 2, 3]])

1 X[:, [True, False, True]] # selecionamos apenas a primeira e última coluna
    array([[1, 3],
           [4, 6]])
```

✓ 1.2) Shape, Reshape e Ravel

Todo vetor, matriz e tensor pode ser redimensionado.

Observe como no tensor abaixo temos $3 \times 2 \times 2 = 12$ elementos. Podemos redimensionar os mesmos para outros tensores com 12 elementos.

```
1 X = np.random.rand(3, 2, 2)
2 print(X.shape, X)

(3, 2, 2) [[[0.02616792 0.53841924]
            [0.06885354 0.3708555 ]]

            [[0.58157638 0.81232297]
            [0.38616519 0.05475409]]

            [[0.36197902 0.7972349 ]
            [0.32928169 0.10106642]]]
```

Podemos redimensionar os elementos como uma matriz 2×6 :

```
1 X_matrix = X.reshape((2, 6))
2 print(X_matrix.shape, X_matrix)

(2, 6) [[0.02616792 0.53841924 0.06885354 0.3708555  0.58157638 0.81232297]
        [0.38616519 0.05475409 0.36197902 0.7972349  0.32928169 0.10106642]]
```

Em Numpy (e PyTorch), podemos fazer com que a biblioteca infira uma dimensão utilizando `-1`:

```
1 X_matrix = X.reshape((2, -1))
2 print(X_matrix.shape, X_matrix) # Note que também temos uma matriz de dimensão: 2x6

(2, 6) [[0.02616792 0.53841924 0.06885354 0.3708555  0.58157638 0.81232297]
        [0.38616519 0.05475409 0.36197902 0.7972349  0.32928169 0.10106642]]
```

Podemos redimensionar os elementos para um outro tensor:

```
1 X_tensor = X.reshape((6, 2, 1))
2 X_tensor.shape

(6, 2, 1)
```

E, por fim, podemos redimensionar os elementos como um vetor, realizando uma operação de `flattening`:

```
1 X_vetor = X.flatten()
2 X_vetor.shape

(12,)
```

Observação: Podemos redimensionar os elementos como um vetor de 3 formas principais:

1. Através da função `flatten`, como visto acima;
2. Através da função `ravel` (presente tanto em Numpy quanto em PyTorch);
3. Através de um `.reshape`, passando como parâmetro o número dos elementos (ou -1).

Os 3 métodos possuem algumas sutilezas que diferenciam um dos outros. Para mais informações consulte o seguinte [link](#).

✓ 2. Tensores em PyTorch

PyTorch é o arcabouço que vamos usar para as nossas tarefas. Assim como o NumPy, o Pytorch é uma biblioteca de processamento vetorial/matricial/tensorial. Operações sobre os tensores do Pytorch possuem sintaxe consideravelmente parecida com operações sobre tensores do NumPy.

O mesmo faz uso de tensores bem similares ao NumPy. Porém, com PyTorch conseguimos fazer uso da GPU.

```
1 import torch
```

✓ 2.1) Casting para o dispositivo correto

Como usaremos processamento vetorial principalmente em GPUs para aprendizado profundo, primeiramente é possível verificar se há uma GPU disponível com o trecho de código abaixo, armazenando os tensores nos dispositivos apropriados.

```
1 if torch.cuda.is_available():
2     device = torch.device('cuda')
3 else:
4     device = torch.device('cpu')
5
6 print(device)

cpu
```

Podemos também realizar essa verificação em uma linha de código (+ pytônico)

```
1 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
2 print(device)

cpu
```

✓ 2.2.) Tensores no Pytorch

Para criar tensores novos, podemos utilizar a função `torch.tensor`, similar à função `numpy.array` da biblioteca Numpy:

```
1 tns = torch.tensor([1, 2, 3, 4, 5, 6])
2 print(tns)

tensor([1, 2, 3, 4, 5, 6])
```

Podemos redimensionar os tensores de maneira similar ao que vimos em Numpy através da função `view`:

```
1 print(tns.view(2, 3))

tensor([[1, 2, 3],
        [4, 5, 6]])
```

Assim como mencionado em Numpy, podemos utilizar `-1` para fazer com que a biblioteca faça uma inferência no formato necessário de acordo com os elementos restantes:

```

1 # Note que obtemos o mesmo tensor anterior
2 print(tns.view(2, -1))

tensor([[1, 2, 3],
        [4, 5, 6]])

```

Observação: Em PyTorch, além de possuímos a função `view`, também possuímos a função `reshape`. Ambas possuem algumas diferenças sutis. Caso queira obter mais informações sobre tais diferenças, acesse o seguinte [link](#). Porém, no geral, iremos trabalhar mais com a função `view` ao desenvolver códigos utilizando PyTorch.

Podemos criar tensores previamente preenchidos por elementos, como 0s através da função `torch.zeros` e 1s através da função `torch.ones`:

```

1 tns_0 = torch.zeros(2, 3) # iniciando um tensor com 0s
2 tns_1 = torch.ones(2, 3) # iniciando um tensor com 1s
3
4 print(tns_0)
5 print(tns_1)

tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])

```

Similar à funções do Numpy, podemos iniciar tensores com valores aleatórios:

```

1 tns_u = torch.rand(2, 3) # valores que seguem uma distribuição uniforme no intervalo [0,1)
2 print(tns_u)
3
4 tns_n = torch.randn(2, 3) # valores que seguem uma distribuição normal N(0,1)
5 print(tns_n)
6
7 tns_perm = torch.randperm(6) # valores que são uma permutação aleatória no intervalo [0, 5]
8 print(tns_perm)

tensor([[0.8309, 0.4705, 0.1576],
        [0.6264, 0.2259, 0.5860]])
tensor([[ -0.9673, -1.5633, -0.1454],
        [ 0.2869,  1.0898,  0.9034]])
tensor([3, 5, 4, 0, 1, 2])

```

Similar à Numpy, podemos realizar operações de soma, multiplicação, entre outras, com tensores:

```

1 print(tns_u)
2 print(tns_n)
3
4 tns_sum = tns_u + tns_n
5 print(tns_sum)

tensor([[0.8309, 0.4705, 0.1576],
        [0.6264, 0.2259, 0.5860]])
tensor([[ -0.9673, -1.5633, -0.1454],
        [ 0.2869,  1.0898,  0.9034]])
tensor([[ -0.1364, -1.0928,  0.0122],
        [ 0.9132,  1.3157,  1.4894]])

```

Similar à Numpy, podemos indexar os tensores da mesma forma apresentada anteriormente:

```

1 print(tns_sum[1, 1]) # Indexando um elemento
2 print(tns_sum[0, :]) # Indexando uma linha (pode também ser tns_sum[0])
3 print(tns_sum[:, 1]) # Indexando uma coluna

tensor(1.3157)
tensor([ -0.1364, -1.0928,  0.0122])
tensor([ -1.0928,  1.3157])

```

Podemos utilizar a função `torch.from_numpy` para converter um tensor de Numpy para PyTorch; ou a função `.numpy()` para converter um tensor de PyTorch para Numpy:

```
1 np_arr = np.random.randn(2, 3)
2 print(np_arr, np_arr.dtype)
3
4 torch_tns = torch.from_numpy(np_arr)
5 print(torch_tns)
6
7 arr = torch_tns.numpy()
8 print(arr)

[[-1.10455944  0.62087257 -1.20315724]
 [-0.0503528  -1.2117737  -1.28111692]] float64
tensor([[[-1.1046,  0.6209, -1.2032],
         [-0.0504, -1.2118, -1.2811]], dtype=torch.float64)
[[-1.10455944  0.62087257 -1.20315724]
 [-0.0503528  -1.2117737  -1.28111692]]
```

Por fim, podemos concatenar dois, ou mais, tensores com a função `torch.cat`:

- O parâmetro `dim` em PyTorch é análogo ao parâmetro `axis` do Numpy. Nele, iremos informar sobre qual dimensão queremos que uma certa operação seja feita. No caso abaixo, ao informarmos a primeira dimensão (0), estamos dizendo para o PyTorch efetuar uma concatenação dos dois tensores a partir da sua primeira dimensão, ou seja, "colando" os tensores verticalmente (no sentido das linhas).

```
1 print((tns_0.shape, tns_1.shape))
2 tns_cat = torch.cat((tns_0, tns_1), dim=0)
3 print(tns_cat, tns_cat.shape)

(torch.Size([2, 3]), torch.Size([2, 3]))
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [1., 1., 1.],
        [1., 1., 1.]]) torch.Size([4, 3])
```

Várias outras operações sobre tensores do Pytorch podem ser vistas nos seguintes tutoriais:

1. <https://jhui.github.io/2018/02/09/PyTorch-Basic-operations/>
2. https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

✓ Conjunto de Problemas 1: Vetorização

Antes de continuar, vamos importar algumas funções que serão utilizadas para testar o resultado dos seus algoritmos.

Estas, vão vir do módulo `testing` do `numpy`.

```
1 from numpy.testing import assert_equal
2 from numpy.testing import assert_almost_equal
3 from numpy.testing import assert_array_almost_equal
```

Seu objetivo é medir a velocidade das operações de álgebra linear para diferentes níveis de vetorização.

1. Construa duas matrizes A e B com entradas aleatórias Gaussianas de tamanho 128×256 .

Dica: Use o módulo `time` para mensurar o tempo da operação.

```
1 A = torch.randn(128, 256)
2 B = torch.randn(128, 256)
```

```

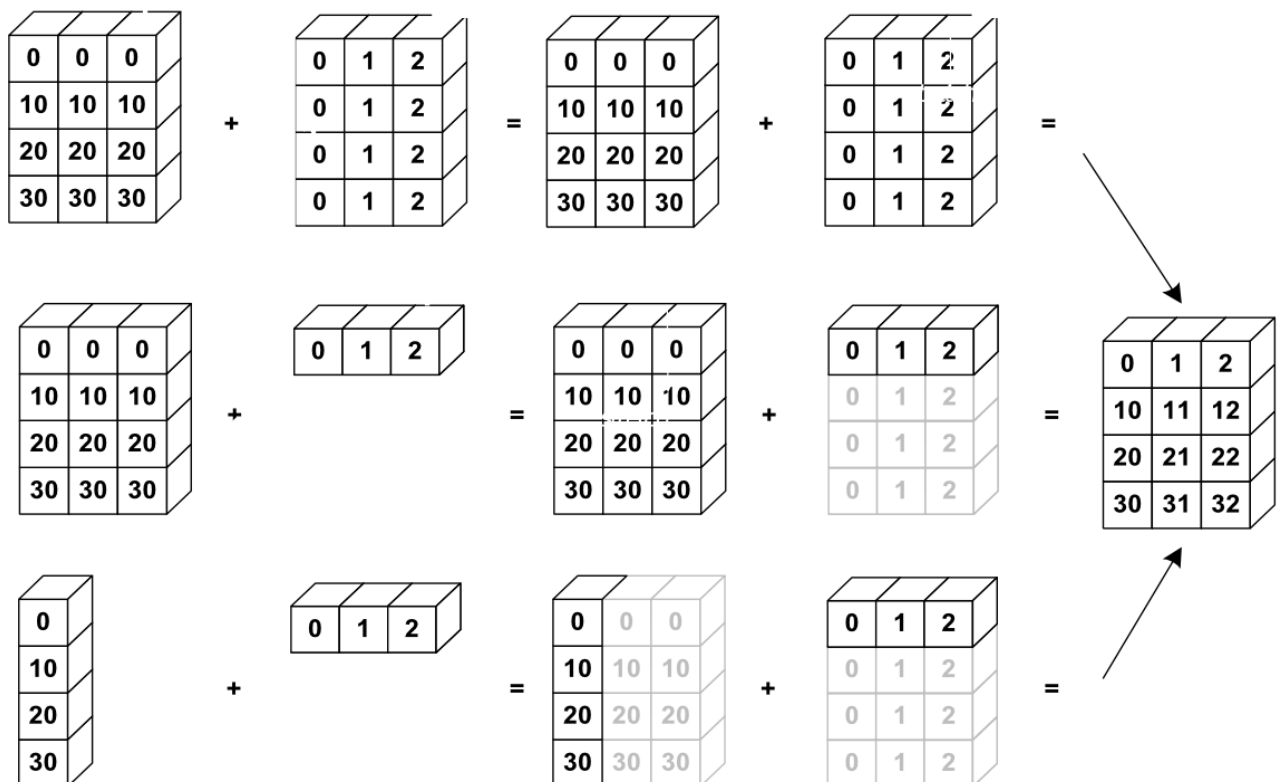
1 # testes, não apague as linhas!!
2 assert_equal((128, 256), A.shape)
3 assert_equal((128, 256), B.shape)
4
5 # A chamada .numpy() converte os vetores em vetores numpy. Útil para testes!
6 Anp = A.numpy()
7 Bnp = B.numpy()
8
9 # testando média e desvio padrão
10 assert_almost_equal(Anp.mean(), 0, decimal=2)
11 assert_almost_equal(Anp.std(ddof=1), 1, decimal=2)
12
13 assert_almost_equal(Bnp.mean(), 0, decimal=2)
14 assert_almost_equal(Bnp.std(ddof=1), 1, decimal=2)

```

2. Calcule $C = AB^T$, tratando A como uma matriz, mas computando o resultado para cada coluna de B . Em outras palavras, realize um produto matricial utilizando um laço `for`! Pare realizar este código, é importante entender o conceito de broadcasting.

Em código numpy e torch, a operação de broadcasting replica linhas e colunas de tensores para realizar operações. Para entender melhor, leia o [documento](#). A figura abaixo exemplifica broadcasting. No geral, as dimensões de arrays casam, as operações são realizadas (primeira linha da figura). Mesmo quando as dimensões não casam, se a última dimensão for compatível é feito a replicação (broadcasting), ver a segunda linha da figura. Por fim, mesmo quando as dimensões não casam mas uma delas é 1 (4x1 + 1x3 na linha 3), é feito broadcasting.

▼



Dica: Você deverá fazer o código em uma linha apenas. Para isso, você vai focar no caso da linha 2 da figura. Multiplique uma linha de A por B . Depois disso, use `.sum(axis=...)` para realizar a soma na dimensão correta.


```

1 import time
2
3 start_time = time.time()
4 C = np.zeros((128, 128))
5 for linha in range(A.shape[0]):
6     C[linha] = (A[linha]*B).sum(axis = 1)
7
8 print('Tempo de execução: ', time.time() - start_time)
9
10

```

Tempo de execução: 0.008784294128417969

```

1 # testes, não apague as linhas!!
2 Cteste = np.matmul(A, B.T) # faz a leitura, realiza operação
3 assert_array_almost_equal(Cteste, C, decimal=3)

```

3. Calcule $C = AB^t$ usando operações matriciais. Ou seja, sem usar nenhum laço. Ao mensurar o tempo, ficou mais rápido?

```

1 start_time = time.time()
2 C = torch.mm(A, B.t())
3 print('Tempo de execução: ', time.time() - start_time)
4

```

Tempo de execução: 0.0007562637329101562

Essa implementação é mais rápida.

```

1 # testes, não apague as linhas!!
2 Cteste = np.matmul(A, B.T) # faz a leitura, realiza operação
3 assert_array_almost_equal(Cteste, C, decimal=3)

```

- Observando o tempo gasto nas duas formas de calcular $C = AB^T$, podemos perceber que ao utilizarmos funções nativas da biblioteca (como `np.matmul` e `torch.mm`), temos uma redução significativa do tempo se compararmos à utilização de um laço `for`.

✓ Conjunto de Problemas 2: Computação eficiente de memória

Crie duas matrizes aleatórias de tamanho 4096×4096 . Chame as mesmas de A e B novamente.

```

1 A = torch.rand(4096, 4096)
2 B = torch.rand(4096, 4096)

```

5. Crie uma função que recebe as matrizes A , B e C , e um número de iterações para atualizar C , de forma que $C = AB^T + C$. Essa função deve, primeiro, calcular a multiplicação de matrizes entre A e B e depois adicionar o valor à C , de acordo com o número de iterações. A mesma deve atualizar C sem alocar memória nova para essa variável.

```

1 def update_c(C, A, B, n_iter=2):
2     D = torch.mm(A, B.t())
3     for _ in range(n_iter):
4         C[:] = D + C

```

```

1 # testes não apague!
2 Ct = torch.zeros(A.shape)
3 Cteste = (Ct + np.matmul(A, B.T))
4 Cteste = (Cteste + np.matmul(A, B.T))
5
6 C = torch.zeros(A.shape)
7 update_c(C, A, B, 2)
8 assert_array_almost_equal(Cteste, C.numpy(), decimal=3)

```

✓ Conjunto de Problemas 3: Programação Diferenciável

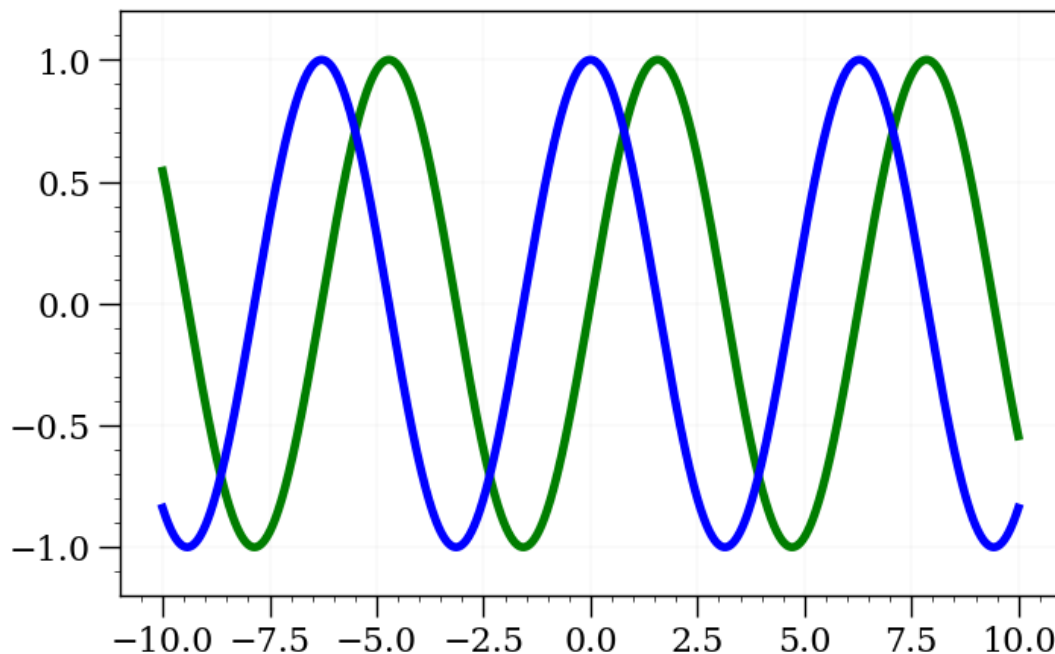
Agora vamos aprender um dos pontos chave de fazer uso de bibliotecas como pytorch/tensorflow/etc, a programação diferenciável. Diferente do exercício que vocês fizeram na mão, usando a biblioteca conseguimos derivar de forma automática. Portanto, observe como o código abaixo deriva a função seno.

```

1 x = np.linspace(-10, 10, 1000)
2 x_torch = torch.tensor(x, requires_grad=True)
3
4 y = torch.sin(x_torch) # seno original
5
6 # Devido ao fato de que o y final não é um escalar precisamos passar o vetor v tal que v é a jacobiana
7 # pela qual vamos multiplicar as jacobianas da variavel node (neste caso, x) conforme descrito em
8 # https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
9 # veremos mais sobre o processo de backproagation, jacobiano, etc no futuro
10 v = torch.ones(x_torch.shape, dtype=torch.double)
11
12 # Como aqui x da origem a y diretamente, nosso v sera apenas um vetor de 1's com a mesma dimensao de x
13 y.backward(v)
14
15 plt.plot(x_torch.detach().numpy(), y.detach().numpy(), 'g', label='sin(x)')
16 plt.plot(x_torch.detach().numpy(), x_torch.grad.numpy(), 'b', label='sin\'(x)')

```

[<matplotlib.lines.Line2D at 0x7dc33e6bbcd0>]



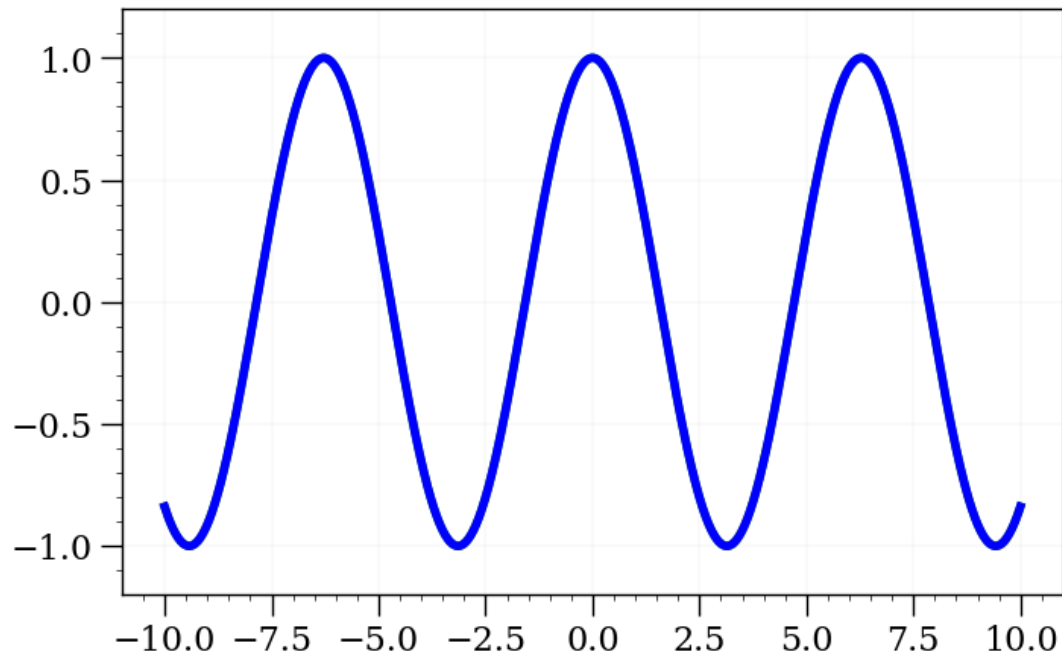
O resultado é a mesma curva da função cosseno! Para entender melhor o autograd, leia a seção respectiva do [pyTorch Blitz](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html).

```

1 plt.plot(x_torch.detach().numpy(), x_torch.grad.numpy(), 'g', label='sin\'(x)')
2 plt.plot(x_torch.detach().numpy(), torch.cos(x_torch).detach().numpy(), 'b', label='cos(x)')

```

[<matplotlib.lines.Line2D at 0x7dc34693a860>]



6. Derive a função logística usando pytorch.

$$f(x) = \frac{1}{1 + e^{-x}}$$

```
1 x = np.linspace(-10, 10, 1000) # Não mude o valor de x!
2 x_torch = torch.from_numpy(x)

1 x_torch = torch.tensor(x, requires_grad=True)
2 y = 1.0 / (1.0 + torch.exp(-x_torch))
3 v = torch.ones(x_torch.shape, dtype=torch.double)
4 y.backward(v)

1 # testes, não apagar
2 y_test = 1.0 / (1 + np.exp(-x))
3 derivada_teste = y_test * (1 - y_test)
4 assert_array_almost_equal(derivada_teste, x_torch.grad.numpy(), decimal=3)
```

A operação *detach* permite quebrar a computação em várias partes. Em particular, isto é útil para aplicar a regra da cadeia.

Suponha que $u = f(x)$ e $z = g(u)$, pela regra da cadeia, temos $\frac{dz}{dx} = \frac{dz}{du} \frac{du}{dx}$. Para calcular $\frac{dz}{du}$, podemos primeiro separar u da computação e, em seguida, chamar `z.backward()` para calcular o primeiro termo.

Observe no caso abaixo como derivamos $u = x^2$. A resposta deve ser $2x$ para cada termo `[0, 1, 2, 3]`.

```
1 x = torch.arange(4, dtype=torch.float)
2 x.requires_grad_(True)
3
4 u = x * x
5 jacobX = torch.ones(x.shape)
6 u.backward(jacobX)
7 x.grad

tensor([0., 2., 4., 6.])
```

Agora vamos fazer $z = u^3$ e computar as derivadas intermediárias.

```

1 x = torch.arange(4, dtype = torch.float)
2 x.requires_grad_(True)
3
4 u = x * x
5 v = u.detach() # u ainda mantém o grafo computacional
6 v.requires_grad_(True)
7 z = v * v * v
8
9 print(z)
10
11 jacobX = torch.ones(x.shape)
12 u.backward(jacobX)
13 x.grad

tensor([ 0.,  1., 64., 729.], grad_fn=<MulBackward0>)
tensor([0., 2., 4., 6.])

```

Acima temos a derivada de x^2 . Abaixo temos a derivada de $g(x^2)$.

```

1 jacobV = torch.ones(v.shape)
2 z.backward(jacobV)
3 v.grad

tensor([ 0.,  3., 48., 243.])

```

7. Agora, sendo $f(x) = 1 + x^2$ e $g(x) = 1 + 7f(x)^4$. Vamos aplicar a regra da cadeia em pytorch

```

1 x = torch.arange(4, dtype = torch.float)
2 x

tensor([0., 1., 2., 3.])

```

```

1 x.requires_grad_(True)
2 u = 1 + x*x
3 jacobX = torch.ones(x.shape)
4 u.backward(jacobX)

```

```

1 # Testando a sua solução
2 # Aqui pode ser meio confuso, mas o gradiente referente à derivada de
3 # f(x) com relação a x, por exemplo, fica armazenada no tensor x, mais especificamente em x.grad
4 assert_array_almost_equal([0, 2, 4, 6], x.grad.numpy())

```

```

1 v = 1 + x * x
2 u = v.detach()
3 u.requires_grad_(True)
4
5 z = 1 + 7 * torch.pow(u, 4)
6
7 jacobX = torch.ones(x.shape)
8 v.backward(jacobX)
9
10 jacobU = torch.ones(u.shape)
11 z.backward(jacobU)
12
13

```

```

1 # Testando a sua solução (caso seja necessário, modifique o nome da sua variável auxiliar para 'u')
2 # Note que agora o gradiente referente à derivada está presente na variável 'u', e não mais em 'x'
3 assert_array_almost_equal([28, 224, 3500, 28000], u.grad.numpy())

```

✓ Conjunto de Problemas 4: Mais Derivadas

Vamos brincar um pouco de derivadas dentro de funções. Dado dois números x e y , implemente a função `log_exp`, que retorna:

$$f(x, y) = -\log\left(\frac{e^x}{e^x + e^y}\right)$$

```
1 def log_exp(x, y):
2     numerador = torch.pow(torch.e, x)
3     denominador = torch.pow(torch.e, x) + torch.pow(torch.e, y)
4     return -torch.log(numerador/denominador)
```

1. Abaixo vamos testar o seu código com algumas entradas simples.

```
1 x, y = torch.tensor([2.0]), torch.tensor([3.0])
2 z = log_exp(x, y)
3 z
# tensor([1.3133])
```

```
1 # Teste. Não apague
2 assert_almost_equal(1.31326175, z.numpy())
```

2. A função a seguir computa $\partial z / \partial x$ e $\partial z / \partial y$ usando autograd.

```
1 # O argumento funcao_forward é uma função python. Será a sua log_exp.
2 # A ideia aqui é deixar claro a ideia de forward e backward propagation, depois
3 # de avaliar a função chamamos backward e temos as derivadas.
4 def grad(funcao_forward, x, y):
5     x.requires_grad_(True)
6     y.requires_grad_(True)
7     z = funcao_forward(x, y)
8     z.backward()
9     return x.grad, y.grad
```

Testando

```
1 x, y = torch.tensor([2.0], dtype = torch.double), torch.tensor([3.0], dtype = torch.double)
2 dx, dy = grad(log_exp, x, y)

1 assert_almost_equal(-0.7310586, dx.numpy())
2 assert_almost_equal(0.7310586, dy.numpy())
```

4. Agora teste com números maiores, algum problema?

```
1 x, y = torch.tensor([400.0]).double(), torch.tensor([800.0]).double()
2 grad(log_exp, x, y)
# (tensor([nan], dtype=torch.float64), tensor([nan], dtype=torch.float64))
```

5. Pense um pouco sobre o motivo do erro acima. Usando as propriedades de logaritmos, é possível fazer uma função mais estável. Abaixo segue a implementação da mesma. O problema aqui é que o exponencial "explode" quando x ou y são muito grandes. Este [link](#) pode ajudar.

```
1 x, y = torch.tensor([400.0], dtype = torch.double), torch.tensor([800.0], dtype = torch.double)
2 def stable_log_exp(x, y):
3     return torch.log(1 + torch.exp(y-x))
4
5 dx, dy = grad(stable_log_exp, x, y)
```

```
1 stable_log_exp(x, y)

   tensor([400.], dtype=torch.float64, grad_fn=<LogBackward0>)

1 # Teste. Não apague
2 assert_equal(-1, dx.numpy())
3 assert_equal(1, dy.numpy())
```